EasyChair Preprint

№ 13948

A Fault Tolerance Mechanism for Hybrid Scientific Workflows

Alberto Mulone, Doriana Medic and Marco Aldinucci

July 12, 2024

A Fault Tolerance Mechanism for Hybrid Scientific Workflows

Alberto Mulone[1], Doriana Medić[1], and Marco Aldinucci[1]

University of Turin, Italy, {name.surname}@unito.it

Abstract. In large distributed systems, failures are a daily event occurring frequently, especially with growing numbers of computation tasks and locations on which they are deployed. The advantage of representing an application with a workflow is the possibility of exploiting Workflow Management System (WMS) features such as portability. A relevant feature that some WMSs supply is reliability. Over recent years, the emergence of hybrid workflows has posed new and intriguing challenges by increasing the possibility of distributing computations involving heterogeneous and independent environments. Consequently, the number of possible points of failure in the execution increased, creating different important challenges that are interesting to study. This paper presents the implementation of a fault tolerance mechanism for hybrid workflows based on the recovery and rollback approach. A representation of the hybrid workflows with the formal framework is provided, together with the experiments demonstrating the functionality of implementing approach.

1 Introduction

In contemporary contexts, the interest in creating and deploying intricate applications across expansive networks of diverse computing architectures is increasing in different scientific domains. Many Workflow Management Systems (WMS) are developed to cope with the demands of applications coming from various fields, from geophysics [12] to bioinformatics [16], simulation of chemical reactions [4] and astrophysics [19]. The necessity to employ different locations to execute a computation has different reasons, such as computational performance or data privacy. The heterogeneity of computing architectures is an increasing trend because new specialised hardware is being developed to execute some tasks performatively. Moreover, data are the most important asset today, and nobody wants to share them for many different reasons, for example data can be sensitive and they are protected for legal reasons. Thus, sometimes, it is impossible to move data, and it is necessary to use them carefully.

Hybrid workflow [8] systems are a possible solution for these requirements in the workflow application. They can span the steps in multiple and heterogeneous environments without sharing memory zones. As said before, hybrid WMSs can move the data to execute a step in a more suitable location based on the step requirements, such as a specific processor, or they can move the computation to minimise the data movements. For example, hybrid workflow

allows the modelling of federated applications, where it cannot be possible to move the data, e.g. orchestrating a federated Deep Neural Network training across different HPC centres [5]. Some of WMS implementing this paradigm are DagOnStar [14], Pegasus [6] and StreamFlow [3].

When workflow applications run on large distributed systems, failure is not a possibility but a daily event whose occurrence increases with the number of activities and actors involved. WMS execution depends on many layers, often perceived as black boxes. It is not guaranteed that the application used to compute steps or the locations used to deploy steps are fault tolerant. Hence, implementing a fault tolerance mechanism directly in WMS to recover the failed step execution is becoming inevitable.

The contribution of this paper is two-sided, on one side providing the implementation of the fault tolerance mechanism for hybrid workflow systems (Sec. 5.1) based on the recovery and rollback techniques (detailed description given in Sec. 3.1), and, on the other side, giving the formalisation of the behaviour of the implemented approach (Sec. 3.2). The formal description of the actions of a hybrid workflow is given through an example (Sec. 4). The fault tolerance mechanism is implemented on StreamFlow, and an experiment with a workflow execution containing the simulated fault behaviour is provided (Sec. 5).

2 Related work

Different types of errors can occur during the workflow execution. They are soft, fail-stop and silent errors [10]. The soft or fail-stop errors cause the failure of the step execution. When a soft error occurs, the input data of the step is still undamaged in the executing locations. Instead, the data is corrupted or lost when a fail-stop error occurs. Finally, detecting the silent errors is more complex since the step terminates with success, but the output data is contaminated (i.e. the step produces the wrong result).

In the literature, various works deal with the mechanisms for error management involving other fields beyond the scientific workflows. Mainly, these mechanisms can be divided into two categories, following the notation of [15], the one dealing with the error when the failure occurs, so-called *reactive methods*, or the one trying to predict and prevent the error, so-called *proactive methods*. Both approaches require additional work to guarantee reliability, such as saving metadata at runtime, duplicating data, increasing the execution time or the number of resources needed compared to an unreliable execution. Therefore, the choice of the method utilised is guided by the type of failure to manage and the availability of resources (time and funding). This paper focuses on reactive methods and different techniques used to recover the failure.

A simple fault tolerance approach is *retry-rollback*, which re-executes the failed step. However, the step can have some dependencies on other steps, which obliges their rollback [7]. In the worst scenario, the workflow is fully re-executed, describing the domino effect [18]. This approach, usually, is coupled with *checkpoint* in which the WMS saves the workflow state and the data at runtime by

applying different heuristics [9]. The recovery of the failure is modelled by bringing the state of the workflow back to the last checkpoint (its valid state) and restarting the execution from there.

Another technique is *replication*, where the WMS executes the step with the same input data multiple times in parallel; hence, if one replica fails, other replicas are still alive, ensuring that the workflow executions continue. This mechanism can also be used to detect silent errors [1].

Finally, the *rescue DAG* approach, implemented in DAGMan [20], saves metadata when a failure occurs and continues to run the workflow until it is possible. In the state where the workflow has only failed and pending steps (waiting for the output of the failed steps), it terminates the execution and creates a new workflow (using saved metadata) containing only the missing steps. This new workflow continues the execution of the original workflow.

3 The fault tolerance mechanism for hybrid workflows

This section describes the considered workflow setting and the fault tolerance implemented, giving a formal syntactical representation of the hybrid workflow and its fault-tolerant mechanism. The workflow is represented as Direct Cyclic Graph (DCG), where the vertices are the steps to be computed and the edges are the dependencies between the steps given by data dependency. It is supposed that the step execution is deterministic since many of the workflow applications we are dealing with are deterministic. Moreover, the step is seen as a black box; thus, the step re-execution is done from the beginning, regardless of its internal state when the failure occurs. The WMS discards the data produced by a failed step because they can be incomplete. Finally, the step can not operate the data in-place, thus input data are still valid if they exist after the failure. The hybrid workflow is obtained by mapping steps to different locations, assuming that the locations are mutually connected following some topology[2].

3.1 The implemented fault tolerance approach

The fault tolerance mechanism employed in this paper is based on the *recovery workflow* approach. Creating a separate workflow aims to retrieve the valid state of the failed steps while the non-affected continue to run their executions. After recovering the failed step, its output is returned to the original workflow. The support of the loops and the concurrent executions between the original workflow and the recovery workflows are the main differences between our solution and the DAGMan approach. The existence of two types of failure-free entities is assumed: WMS, in this paper called *driver* or *driver location*, and all locations on which the initial dataset is stored. In this way, it is guaranteed that the initial input to re-execute the whole workflow (if necessary) will not be lost.

The presented implementation manages two types of errors, soft and fail-stop errors, that occur during step execution. Specifically, during the implementation,

the considered failures are the application failure (e.g., a segmentation fault occurs) and the location failure (e.g., the HPC system has extraordinary maintenance while a step is running). In the case of application failure, it is enough to re-execute the step on the location[1]. Instead, in the case of location failure, all the data on the location can be lost when it has ephemeral memory (e.g., in the Kubernetes Pods without a permanent volume). In this case, there are two possible scenarios. It is possible to copy the input data to the failed location, and the same strategy of application failure can be applied if the data are available in another location. Otherwise, if the data is not present in any location, it is necessary to roll back and recover the steps that produced the lost data.

As mentioned in Sec. 2, the fault tolerance mechanism introduces some overhead. In this approach, the overhead is given by metadata collection and increased workflow execution time. The metadata tracks the step data produced, the required inputs, and the location where the data resides. These metadata are stored by the driver; however, the data are across different locations. Thus, the driver saves a token that represents the data. They are collected as a graph, called *provenance graph*, where the vertices are the data information and the edges are the dependencies between the data. In the implementation, the steps to rollback are decided through the bottom-up navigation of the provenance graph with a Breadth-First Search (BFS) visit. The starting data of the navigation are the input data of the failed step. If the data of the provenance graph vertex is available in some locations, it becomes an input of the recovery workflow. Otherwise, it is necessary to visit the parent vertices of the current vertex.

3.2 Syntactical representation

This section gives a hint on a formalisation of the fault tolerance mechanism for the hybrid workflows. The workflow description follows the idea presented in [2], while the formalisation is inspired by the distributed π-calculus approach [11] to model location aware workflows [13] with a more elaborated framework to be able to cover the recovery part of the workflow. As in [2], the information about the location is recorded into the *location configuration*, in this case, containing the name of the location l, the set of the data and messages saved at the location l (denoted by D_l) and the trace t of the actions to be performed on the location l. Another addition respect to the semantics given in [2] is the modelling of the *driver location*, which contains the additional elements, saved into the set D_{l_d}, the trace of the whole workflow structure $t_{\mathtt{W}}$ and the mapping function $\mathcal{M}(s)$ that maps the step s to the location on which it is deployed. The driver location has a global knowledge of the workflow and its execution. It orchestrates the execution of the steps: knows when the data is produced or received, manages all recovery actions, and so on.

The traces are built by the following actions: *execution* of the step s, denoted as $\mathtt{exec}(s, I, O)$ recording sets of input and output data of the step s; *transfer*,

[1] It is possible also to change location, but other policies are necessary, such as copying the input data to a new location if it is needed

denoted by couple of prefixes $\texttt{tran}(v, l_2)$ and $\underline{\texttt{tran}}(v, l_1)$ modeling the transfer of the value v from location l_1 to the location l_2. The value v could be the data d or different messages: $m_{d,l}$, sent from location l to the driver, signalling that the data d is produced on the location l; m_s, sent from the driver to indicate that the step s can be executed; ok_d and ok_{m_s}, sent to the driver to confirm the reception of the corresponding data/message; $err(s)$, sent to the driver location indicating that the step s failed (soft error) and the data to re-execute it are still present on the location; $err(D, l)$, transferred to the driver indicating that the data contained in the set D is missing on the location l due to the location fails (fail-stop error); $err(d, l)$ and $err(m_s)$ indicating that the data and the message are not received on the location l due to the transfer failure. Finally, $\texttt{rec}(x)$ is the recovery of the failed element x. This element can be the step s, in which the application fails, the lost data, or the value not received due to transfer failure.

The basic actions are composed in the trace by applying different operators, sequential execution ., parallel composition $\mid$ and the choice operator $+$. Differently from the classic process algebra, the prefix representing the executed step is not discarded but annotated with the pointer $^{\triangleright}$. For instance, considering the trace $t = \texttt{exec}(s, I, O).\texttt{tran}(v, l_2)$, in the classical case, after the execution of the step, the obtained trace is $t' = \texttt{tran}(v, l_2)$, representing only the actions to be executed in the future. In our case, the initial trace t is annotated with the pointer, indicating the state of the execution; therefore, the trace t becomes $^{\triangleright}\texttt{exec}(s, I, O).\texttt{tran}(v, l_2)$, and after the execution of the step, the obtained trace is $t'' = \texttt{exec}(s, I, O).^{\triangleright}\texttt{tran}(v, l_2)$ indicating that the next action to be performed is transfer of the value v. Formally:

Definition 1. *The grammar defines the syntax of a workflow system* $\texttt{W}$*:*

$$
\begin{aligned}
\texttt{W} &::= \langle l, D_l, t\rangle \parallel (\texttt{W}_1 \mid \texttt{W}_2)\\
t &::= \mu \parallel t_1.t_2 \parallel (t_1 \mid t_2) \parallel (t_1 + t_2) \parallel {}^{\triangleright}\, t \parallel \mathbf{0}\\
\mu &::= \texttt{exec}(s, I, O) \parallel \texttt{tran}(v, l_2) \parallel \underline{\texttt{tran}}(v, l_1) \parallel \texttt{rec}(x)\\
v &::= d \parallel m_s \parallel m_{d,l} \parallel ok_{m_s} \parallel ok_d \parallel err(x)\\
x &::= s \parallel D, l \parallel d, l \parallel m_s, l
\end{aligned}
$$

The confirmation of the data/message received can be modelled using the syntax of Def. 1 in the following way:

$$\mathbf{conf}(x, l) = \texttt{tran}(ok_x, l_d) + \texttt{tran}(err(x, l), l_d) \text{ where } x = m_s \ \vee \ x = d.$$

For instance, location l confirms to the driver location the reception of the data d (denoted as $\mathbf{conf}(x, l)$) by transferring to it or the message ok_d, meaning that the data is received successfully, or $err(d, l)$ indicating that the data is not received on the location l.

Notation. We introduce some simplifications of the notation, when possible, to improve the paper's readability. Therefore, we write $\texttt{exec}(s)$, instead of $\texttt{exec}(s, I, O)$ and $\underline{\texttt{tran}}(x, l).\mathbf{conf}$ instead of $\underline{\texttt{tran}}(x, l).\mathbf{conf}(ok_x, l')$ since from $\underline{\texttt{tran}}(x, l)$ is clear what is the message which reception is necessary to confirm. To simplify the traces of the step (actions necessary to execute the step and transfer the data to the succeeding computations), the trace regarding the step s can be

modelled as: input actions, divided in input of the message m_s and the input of necessary data (denoted as trace $t_{I(s)} = \prod_j \underline{\texttt{tran}}(d_j, l_h).\mathbf{conf}(ok_{d_j}, l_h)$); execution of the step; actions signalling to the driver about the outcome of the step (confirming the produced data or indicating an error in the execution); and output actions transferring data to the targeting locations ($t_{O(s)}$). Considering this, the trace of any step s can be represented with:

$$t(s) = (\underline{\texttt{tran}}(m_s, l_d).\mathbf{conf} \mid t_{I(s)}).\texttt{exec}(s).(\prod_i (\texttt{tran}(m_{d_i,l}, l_d) + \texttt{tran}(err(x), l_d))).t_{Out(s)}$$

To simplify, the notation $t(s_{\setminus t_{I(s)}})$ denotes the trace $t(s)$ without the input data part ($t_{I(s)}$). The trace on the driver location which represents the orchestration of the step s mapped to the location l is complementary to the $t(s)$ and it is modelled as:

$$t_d(s) = (\texttt{tran}(m_s, l).\underline{\mathbf{conf}} \mid t_{d_{I(s)}}).(\prod_i (\underline{\texttt{tran}}(m_{d_i,l}, l) + \underline{\texttt{tran}}(err(x), l)))$$

where $t_{d_{I(s)}} = \prod_j \texttt{tran}(d_j, l).\mathbf{conf} \mid \prod_h \mathbf{conf}(ok_{d_h}, l)$ means that the driver sends data d_j to the step s and receives the confirmations ok_{d_h} of the data received on location l.

4 Formalisation through the example

Representation of the hybrid workflows in the formal language defined with Def. 1 is given by modelling the workflow provided in Fig. 1. For the sake of paper readability, we show the mechanism by concentrating on the specific action, while the semantics rules ([17, Fig. 5 and 6]), extensive definition of the workflow behaviour and illustration of the recovery mechanism are provided in App. [17]. It is assumed that the necessary contextual rules, together with the structural congruence rules like commutativity of the parallel and choice operator, neutral element for the sequence and parallel operator (element $\mathbf{0}$), and so on, are holding.

The initial state of the workflow in Fig. 1 can be represented using the syntax of Def. 1 in the following way:

$$\mathtt{W} = \langle l_d, D_{l_d}, {}^{\triangleright} t_d, t_{\mathtt{W}} \rangle \mid \prod_{i=1}^{4} \langle l_i, \emptyset, \emptyset \rangle$$

where $D_{l_d} = \{t_{\mathtt{W}}, \mathcal{M}(s_i), d, m_{s_i}\}$, $i = 1, \ldots, 4$ and $t_{\mathtt{W}} = \prod_{i=1}^{4} t(s_i)$. Trace t_d consists of four subtraces $t_d(s_i)$, executing on the driver location and orchestrating the executions of the four steps s_i. The trace $t_{\mathtt{W}}$ comprises the four traces $t(s_i)$ representing the actions to perform on the locations on which the steps s_i are deployed. For instance, $t(s_3)$ denotes the inputs of the data d_2 and message m_{s_3}; the confirmation of the reception, the execution of the step which produces the new data d_4 or complementing error, then communicating to the driver about the value produced and when the data is produced, finally, transferring the data to the next step/location (s_4/l_1). Formally:

$$\begin{aligned} t(s_3) = &(\underline{\texttt{tran}}(d_2, l_1).\mathbf{conf}(ok_{d_2}, l_2) \mid \underline{\texttt{tran}}(m_{s_3}, l_d).\mathbf{conf}). \\ &\texttt{exec}(s_3).(\texttt{tran}(m_{d_4,l_2}, l_d) + \texttt{tran}(err(x), l_d)).\texttt{tran}(d_4, l_1) \end{aligned}$$

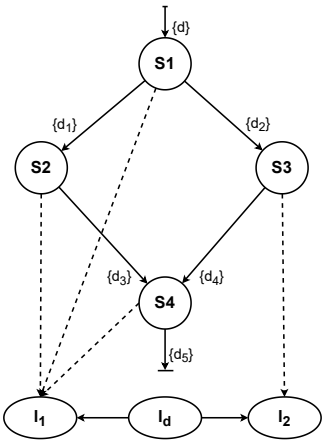

Fig. 1. The hybrid workflow consisting of four steps (represented as circles), connected in the following way: step s_1 produces data d_1 and d_2 necessary for the execution of the steps s_2 and s_3, respectively. The resulting data of those two steps trigger the execution of the step s_4. The steps s_1, s_2 and s_4 are deployed on the location l_1, while step s_3 is mapped on the location l_2. The driver location contains the initial input data of the workflow, and it orchestrates the executions.

The corresponding trace on the driver location $t_d(s_3)$ would be:

$$t_d(s_3) = \texttt{init}(t(s_3), l_2).(\texttt{tran}(m_{s_3}, l_2).\underline{\textbf{conf}} \mid \underline{\textbf{conf}}(ok_{d_2}, l_2)).(\underline{\texttt{tran}}(m_{d_4,l_2}, l_2) + \underline{\texttt{tran}}(err(x), l_2))$$

The driver first initialises the trace of the step on the deploying location, then receives the confirmation that necessary data is received on the location and triggers the step execution. At the end, it receives the produced data or the obtained error. A full representation of the traces can be found in App. [17].

The following text shows how some actions work, by using the traces $t(s_3)$ and $t_d(s_3)$. In particular are shown the transfer of the data, successful step execution and unsuccessful execution with recovery mechanism. To simplify the presentation and highlight the executing action, $\mathbf{T}[\ ]$ denotes the context of the executing action. Therefore, the traces $t(s_3)$ and $t_d(s_3)$, which show the transfer of the message m_s from the driver location to the location l_2, can be written as:

$$t(s_3) = \mathbf{T}[\ ^{\triangleright}\underline{\texttt{tran}}(m_{s_3}, l_d)\ ] \qquad \text{and} \qquad t_d(s_3) = \mathbf{T}'[\ ^{\triangleright}\texttt{tran}(m_{s_3}, l_2)]$$

Componing the traces and the locations, there is:

$$\langle l_d, D_{l_d}, \mathbf{T}[\ ^{\triangleright}\texttt{tran}(m_{s_3}, l_2)]\rangle \mid \langle l_2, D_{l_2}, \mathbf{T}'[\ ^{\triangleright}\underline{\texttt{tran}}(m_{s_3}, l_d)]\rangle \rightarrow$$
$$\langle l_d, D_{l_d}, \mathbf{T}[\texttt{tran}(m_{s_3}, l_2)^{\triangleright}]\rangle \mid \langle l_2, D_{l_2} \cup \{m_{s_3}, ok_{m_{s_3}}\}, \mathbf{T}'[\underline{\texttt{tran}}(m_{s_3}, l_d)^{\triangleright}]\rangle$$

When all necessary elements are on the location l_2, the step s_3 can be executed successfully:

$$\langle l_2, D_{l_2}, \mathbf{T}''[\ ^{\triangleright}\texttt{exec}(s_3)]\rangle \rightarrow \langle l_2, (D_{l_2} \setminus m_{s_3}) \cup \{m_{d_4,l_2}, d_4\}, \mathbf{T}''[\texttt{exec}(s_3)^{\triangleright}]\rangle$$

Considering the failure of a step, for instance, the computation of the step has failed and the data is still present on the location l_2, then the computation would be:

$$\langle l_2, D_{l_2}, \mathbf{T}''[\ ^{\triangleright}\texttt{exec}(s_3)]\rangle \rightarrow \langle l_2, (D_{l_2} \setminus m_{s_3}) \cup \{err(s_3)\}, \mathbf{T}''[\texttt{exec}(s_3)^{\triangleright}]\rangle$$

To recover the error, the failure is communicated to the driver location by adding the $\texttt{rec}(s_3)$ trace to the driver execution trace:

$$\langle l_2, D_{l_2}, \mathbf{T}'''[^{\triangleright}\texttt{tran}(err(s_3), l_d)]\rangle \mid \left\langle l_d, D_{l_d}, \mathbf{T}^{\mathbf{iv}}[^{\triangleright}\underline{\texttt{tran}}(err(s_3), l_2)]\right\rangle \rightarrow$$
$$\langle l_2, D_{l_2} \setminus err(s_3), \mathbf{T}'''[\texttt{tran}(err(s_3), l_d)^{\triangleright}]\rangle \mid$$
$$\left\langle l_d, D_{l_d} \cup err(s_3), \mathbf{T}^{\mathbf{iv}}[\underline{\texttt{tran}}(err(s_3), l_2)^{\triangleright}] \mid {}^{\triangleright}\texttt{rec}(s_3)\right\rangle$$

By executing the recover action, the driver location is coordinating the necessary computation by repeating the step execution without transferring the data since it is already present on the location:

$$\langle l_d, D_{l_d}, t'_d \mid {}^{\triangleright}\texttt{rec}(s_3)\rangle \mid \langle l_2, D_{l_2}, t'(s_3)\rangle \rightarrow$$
$$\left\langle l_d, D_{l_d} \setminus err(s_3), t'_d \mid \texttt{rec}(s_3)^{\triangleright} \mid {}^{\triangleright}t_d(s \setminus t_{d_{I(s_3)}})\right\rangle \langle l_2, D_{l_2}, \mathbf{0} \mid {}^{\triangleright}t(s_3 \setminus t_{I(s_3)})\rangle$$

where $t(s_3 \setminus t_{I(s_3)})$ (similar for the trace $t_d(s \setminus t_{d_{I(s_3)}})$) is:

$$t(s_3 \setminus t_{I(s_3)}) = \underline{\texttt{tran}}(m_{s_3}, l_d).\mathbf{conf}.\texttt{exec}(s_3).$$
$$(\texttt{tran}(m_{d_4,l_2}, l_d) + \texttt{tran}(err(x), l_d)).\texttt{tran}(d_4, l_1)$$

5 Implementation and experiments

5.1 Implementation

We implemented this mechanism on StreamFlow, a hybrid WMS based on the open standard Common Workflow Language (CWL). It has a well-defined module structure, allowing the community to develop some plugins to customise the default StreamFlow features, such as a new scheduling policy.

The fault tolerance implementation has some features and optimisations that are not currently included in the semantics. A feature of StreamFlow that is supported in our implementation is the possibility of having loops in the workflow. Loops change nothing in our idea; indeed, the *provenance graph* is always a Direct Acyclic Graph (DAG) because the iterations already executed are unfolded in the provenance. Furthermore, thanks to creating a new workflow, the loop management is responsible for the workflow engine, in this case, the StreamFlow driver, which already supports loops. However, it was necessary to introduce a mechanism to stop the recovery workflow and its loop immediately after the failed step was recovered. An optimisation regarding the semantics is synchronising recovery workflows that want to roll back the same step concurrently. The implementation introduces a dependency between the steps across different recovery workflows. The driver manages this synchronisation.

5.2 Experiment environment

We developed an application and encapsulated it in a Docker image[2]. The application has some hyper-parameters to raise the failures. It is possible to choose

[2] `https://hub.docker.com/repository/docker/mul8/sf-failure/general`

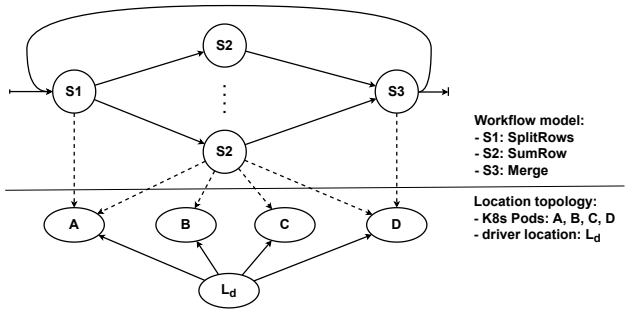

Fig. 2. Workflow model presents a loop of 3 steps where the S2 (i.e. *SumRow*) step has multiple instances. These steps are mapped on the A, B, C and D locations.

the probability and the type of failure occurrence. The failure type can be *tool*, i.e. soft error, or *resource*, i.e. fail-stop error. The latter simulates a resource preemption or a shutdown of the location for any reason. The application forces the Docker container to terminate with error to simulate the fail-stop error.

In the experiments, we ran the StreamFlow driver on our local machine, which also contains the dataset. Instead, we configured a virtualised Kubernetes (K8s) cluster for the remote locations. The K8s cluster has 3 control plane nodes and 4 worker nodes. Each node has 4 virtual CPUs and 8GB of memory. In our experiment, we used 4 Pods and each one uses our Docker image. When the Docker container exits with an error, it also fails the Pod, and K8s will restart it. All data in the Pod will be lost because it does not have a persistent volume.

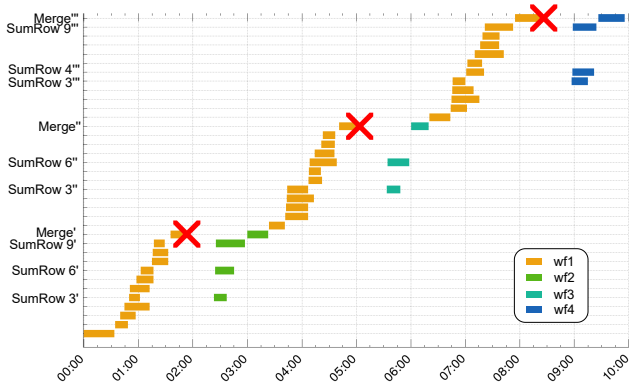

Fig. 3. Execution of the workflow in Fig. 2. Some step names are omitted for the sake of readability. The execution is represented in minutes and seconds.

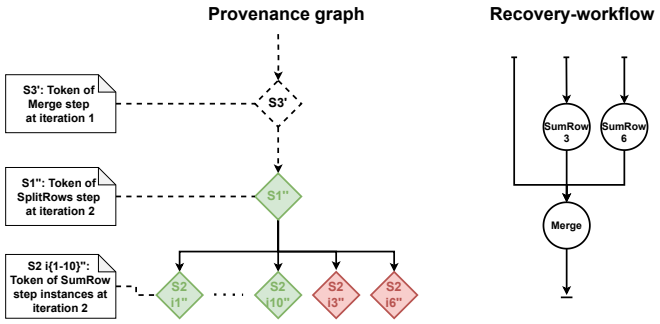

Fig. 4. Left: the visited *provenance graph*. Right: the created recovery workflow.

5.3 Experiment

Fig. 2 shows the workflow configuration. The steps are enclosed in a loop; the *SumRow* step has different instances based on how much output data the *SplitRows* generate. The exit condition of the loop is to reach the i-th iteration. In this experiment, we set it at three iterations; moreover, the *SplitRows* generates 10 data, so there will be 10 *SumRow* step instances. We used four locations, i.e. K8s Pods, named A, B, C and D. The *SplitRow* step is mapped on the location A and the *Merge* step is mapped on the location D. *SumRow* instances are mapped on all the locations, and each is scheduled on the first available location. Moreover, we set the location failure only in the *Merge* step with 50% of probability.

Fig. 3 shows an example of execution. The bars represent the step execution, and the cross represents the failure of the step. The wf1 is the original workflow. Instead, the wf2-4 are recovery workflows. In this execution, the container in *Merge* step throws the location failure at each iteration. The recovery mechanism creates a recovery workflow with the appropriate steps to rollback (e.g., the wf3, which is the teal-colored bars in Fig. 3). For example, Fig. 3 shows the *Merge* step fails at the second iteration, called *Merge"*, while Fig. 4 shows the recovery mechanism of the failure. In the *provenance graph*, the green diamonds are the available data in some locations, and the red diamonds are lost data. The visit of the *provenance graph* starts from its input data; in this case, there are only two lost data. In particular, these lost data are produced by *SumRow 3"* and *6"*. We lose these data because these two steps have been scheduled and executed on the same location of *Merge"*, i.e. location D, and the data were not copied to other locations. The recovery workflow is created with 3 steps. The available data are in the input of the *Merge* step, whileas, the *SumRow* 3" and 6" steps are re-executed. In this experiment, we do not use fully the potential of hybrid workflows because we used only homogeneous locations. However, the approach does not change. If a location fails but becomes available again, as in the case of K8s Pod, it can be used to re-execute the step. Otherwise, it is possible to apply a scheduling policy that changes the location to retry the step execution. In the implementation, there is a delay, choosen by the user, before retrying to

communicate with a failed location; indeed, in the experiment, the delay was set to 20 seconds. These 20 seconds of delay can be seen in Fig. 3; in fact, there are idle times between the failure steps and the first step of the recovery workflows.

6 Conclusions

Given the escalating computational demands of scientific applications, their execution time remains a significant challenge. Therefore, a robust system that safeguards against the loss of hours or days of computation is crucial. This underscores the importance of WMSs and their role in ensuring such applications' reliability. Moreover, the hybrid workflow paradigm, a relatively new and versatile approach, has gained traction recently. In this work, we show a fault tolerance mechanism for hybrid workflows. We also model the idea with semantics, showing an example of execution. Then, we implement the concept and show a workflow with different patterns, such as loops and multi-instances.

One key advantage of our fault tolerance mechanism is the absence of complex logic in the workflow engine to restore previous internal states or undo some actions. Instead, it is just necessary to synchronise different workflow executions. Finally, we can retrieve the successful steps that produce the workflow output by navigating the provenance graph, which is built across different workflows. Future work involves evaluating implementation overhead, especially with more complex configurations using real-case workflows and different numbers of locations. It also involves combining the retry-rollback with checkpointing the data in safe locations. Other future work involves aligning semantics with the implementation describing synchronisation and loops and extending both with nondeterministic workflows.

Acknowledgments. This work was supported by: the Spoke 1 "FutureHPC & BigData" of ICSC - Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing, funded by European Union - NextGenerationEU; the EUPEX EU's Horizon 2020 JTI-EuroHPC research and innovation programme project under grant agreement No 101033975.

References

1. Benoit, A., et al.: Identifying the right replication level to detect and correct silent errors at scale. In: Proceedings of the 2017 Workshop on Fault-Tolerance for HPC at Extreme Scale. p. 31–38. FTXS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3086157.3086162
2. Colonnelli, I.: Workflow models for heterogeneous distributed systems. Ph.D. thesis, Università degli Studi di Torino (Nov 2022). https://doi.org/10.5281/zenodo.7273357
3. Colonnelli, I., et al.: StreamFlow: cross-breeding cloud with HPC. IEEE Transactions on Emerging Topics in Computing **9**(4), 1723–1737 (2021). https://doi.org/10.1109/TETC.2020.3019202

4. Colonnelli, I., et al.: Distributed workflows with jupyter. Future Generation Computer Systems **128**, 282–298 (2022). https://doi.org/10.1016/j.future.2021.10.007
5. Colonnelli, I., et al.: Federated learning meets HPC and cloud. In: Astrophysics and Space Science Proceedings. vol. 60, pp. 193–199. Springer, Catania, Italy (2023). https://doi.org/10.1007/978-3-031-34167-0_39
6. Deelman, E., et al.: The evolution of the Pegasus workflow management software. Computing in Science and Engineering **21**(4), 22–36 (2019). https://doi.org/10.1109/MCSE.2019.2919690
7. Elnozahy, E.N.M., et al.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. **34**(3), 375–408 (sep 2002). https://doi.org/10.1145/568522.568525
8. Ferreira da Silva, R., et al.: A characterization of workflow management systems for extreme-scale applications. Future Generation Computer Systems **75**, 228–238 (2017). https://doi.org/10.1016/j.future.2017.02.026
9. Han, L., et al.: Checkpointing workflows for fail-stop errors. IEEE Transactions on Computers **67**(8), 1105–1120 (2018). https://doi.org/10.1109/TC.2018.2801300
10. Han, L., et al.: A generic approach to scheduling and checkpointing workflows. The International Journal of High Performance Computing Applications **33**(6), 1255–1274 (2019). https://doi.org/10.1177/1094342019866891
11. Hennessy, M.: A distributed Pi-calculus. Cambridge University Press (2007). https://doi.org/10.1017/CBO9780511611063
12. Klampanos, I.A., et al.: Dare platform a developer-friendly and self-optimising workflows-as-a-service framework for e-science on the cloud. Journal of Open Source Software **5**(54), 2664 (2020). https://doi.org/10.21105/joss.02664
13. Medic, D., et al.: Towards formal model for location aware workflows. In: 47th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2023, Torino, Italy, June 26-30, 2023. pp. 1864–1869. IEEE (2023). https://doi.org/10.1109/COMPSAC57700.2023.00289
14. Montella, R., et al.: Dagon*: Executing direct acyclic graphs as parallel jobs on anything. In: 2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS). pp. 64–73 (2018). https://doi.org/10.1109/WORKS.2018.00012
15. Mukwevho, M.A., et al.: Toward a smart cloud: A review of fault-tolerance methods in cloud systems. IEEE Transactions on Services Computing **14**(2), 589–605 (2021). https://doi.org/10.1109/TSC.2018.2816644
16. Mulone, A., et al.: Porting the Variant Calling Pipeline for NGS data in cloud-HPC environment. In: 2023 IEEE 47th Annual Computers, Software, and Applications Conference. pp. 1858–1863 (Jun 2023). https://doi.org/10.1109/COMPSAC57700.2023.00288
17. Mulone, A., et al.: A fault tolerance mechanism for hybrid scientific workflows (2024), https://arxiv.org/abs/2407.05337
18. Randell, B.: System structure for software fault tolerance. SIGPLAN Not. **10**(6), 437–449 (apr 1975). https://doi.org/10.1145/390016.808467
19. Sciacca, E., et al.: An integrated visualization environment for the virtual observatory: Current status and future directions. Astronomy and Computing **11**, 146–154 (2015). https://doi.org/10.1016/j.ascom.2015.01.006, the Virtual Observatory: II
20. Thain, D., et al.: Distributed computing in practice: the condor experience. Concurrency - Practice and Experience **17**, 323–356 (02 2005). https://doi.org/10.1002/cpe.938